Understanding the Backward Slices of Performance Degrading Instructions

Craig Zilles and Guri Sohi

University of Wisconsin - Madison

International Symposium on Computer Architecture

June, 2000
Motivation

Processors achieve only a fraction of peak performance on many programs

- Performance Degrading Events (PDE)
  - branch mispredictions
  - cache misses

Larger caches and predictors

- handle easy cases
- concentrates PDE’s to a fraction of “problem” static instructions
  - Uncorrelated (data-dependent) branches
  - Hash table lookups and pointer chasing
Motivation, cont.

**Program behavior is deterministic**

→

**Build predictors which use the program!**

Pre-execution

Pre-executed sub-program feeds prediction for branch fetched by the main thread

Pre-fetch memory similarly

Only pre-execute instructions which defy normal predictors

Avoid misprediction
The effectiveness of pre-execution is determined by the sub-program:

- Sub-program must enable faster execution of the problem instruction.
- Sub-program size determines “overhead”.

How can we build sub-programs to minimize their size while maintaining accuracy?
Overview

• Motivation
• Program Slicing
• Experiment Overview
• Methodology
• Conservative Slices
• Example Speculative Optimizations
• Additional Observations
• Conclusions
**Program Slicing**

THE SUB-PROGRAM INCLUDES ONLY THE SUBSET OF INSTRUCTIONS WHICH CAN INFLUENCE THE PROBLEM INSTRUCTION.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Register(s)</th>
<th>Destination Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lda</code></td>
<td><code>r8, -8432(r29)</code></td>
<td></td>
</tr>
<tr>
<td><code>cmoveq</code></td>
<td><code>r18, r1, r0</code></td>
<td></td>
</tr>
<tr>
<td><code>ldl</code></td>
<td><code>r1, -19952(r29)</code></td>
<td></td>
</tr>
<tr>
<td><code>s4addq</code></td>
<td><code>r16, r8, r8</code></td>
<td></td>
</tr>
<tr>
<td><code>stl</code></td>
<td><code>r31, 0(r8)</code></td>
<td></td>
</tr>
<tr>
<td><code>and</code></td>
<td><code>r4, r5, r5</code></td>
<td></td>
</tr>
<tr>
<td><code>s1l</code></td>
<td><code>r5, 4, r5</code></td>
<td></td>
</tr>
<tr>
<td><code>ldq</code></td>
<td><code>r23, -19408(r29)</code></td>
<td></td>
</tr>
<tr>
<td><code>ldl</code></td>
<td><code>r27, -19944(r29)</code></td>
<td></td>
</tr>
<tr>
<td><code>addl</code></td>
<td><code>r1, 1, r1</code></td>
<td></td>
</tr>
<tr>
<td><code>addq</code></td>
<td><code>r0, r5, r5</code></td>
<td></td>
</tr>
<tr>
<td><code>bis</code></td>
<td><code>r31, r31, r0</code></td>
<td></td>
</tr>
<tr>
<td><code>stl</code></td>
<td><code>r1, -19952(r29)</code></td>
<td></td>
</tr>
<tr>
<td><code>ldq</code></td>
<td><code>r7, 8(r5)</code></td>
<td></td>
</tr>
</tbody>
</table>

**BACKWARD SLICE**

- `cmoveq` `r18, r1, r0`
- `and` `r4, r5, r5`
- `sll` `r5, 4, r5`
- `addq` `r0, r5, r5`
- `ldq` `r7, 8(r5)`

follow dependences backward from criterion instruction

both data and control

Criterion Instruction
Experiment Overview

Initial characterization of slices

Categorization of instructions in the slice

• largest contributors:
  o control and memory dependence resolution
  o NOT data flow

Exploiting well-known phenomena to reduce slice size

• highly-biased branches
• stability of memory dependences
Methodology

- **Spec95** integer benchmarks
  - Alpha architecture, optimized -O4

- Profiled to identified “problem” static instructions
  - Frequently caused mispredictions or cache misses

- Generated **assembly-level** slices
  - Looked at the 512 dynamic instructions leading to the criterion
  - Removed NOPS, and SP/GP computation dependences

![Graph showing cumulative slice size against distance (in dynamic instructions)]
**Conservative Slices**

Programs have ambiguous memory dependences and complex control flow → conservatively constructed slices can be large

- 50% of program necessary to compute all store addresses
- 80% of program necessary to resolve all branches

**Solution:** exploit the fact that slices only provide hints

- construct speculative slices
- assume common-case behavior
  - profiling is required to detect the common-case

**Two example optimizations:**

- both targeting ambiguous memory dependences
Speculative Optimizations

A CONSERVATIVE SLICE MUST COMPUTE THE ADDRESS FOR EVERY STORE WHICH COULD ALIAS WITH A LOAD IN THE SLICE.

THE SET OF MEMORY DEPENDENCIES ACTUALLY REALIZED IS A SUB-SET OF THOSE WHICH ARE POSSIBLE.

- Profile to identify the store sets
- Only compute store addresses for these stores

Slices built using store sets approximate size of oracle slices.
When memory dependences exist, often the load communicates with the most recent store from its store set. 

• much like communication through registers

Exploit to reduce slice size:

1. Assume a communication pattern (imprecise transformation)
2. Remove load and store from the slice

\[
\begin{align*}
  & \text{STORE } R9 \rightarrow 0(R17) \\
  & \text{ADD } R10, R11 \rightarrow R12 \\
  & \text{STORE } R12 \rightarrow 0(R18) \\
  & \text{LOAD } 0(R19) \rightarrow R13 \\
  & \text{ADD } R13, R14 \rightarrow R15 \\
  & \ldots \\
  & \text{STORE } R9 \rightarrow 0(R17) \\
  & \text{ADD } R10, R11 \rightarrow R12 \\
  & \text{STORE } R12 \rightarrow 0(R18) \\
  & \text{LOAD } 0(R19) \rightarrow R13 \\
  & \text{ADD } R12, R14 \rightarrow R15 \\
  & \ldots
\end{align*}
\]
**Speculative Optimizations, cont.**

**Static loads are highly biased with respect to this behavior**

- Mis-speculation can be avoided
- Easily profiled to classify the dependences

*Removing such load-store pairs can further reduce slice size with little affect on accuracy*

*Reduced address sub-slice to 1/4 of conservative size*
Additional Observations

Often data dependences are clustered near criterion
  - possibly influences pre-execution mechanisms

Not uncommon for slices to overlap
  - create a single “multi-slice”

Slices are bursty
  - due to program structure

- Diagram with cumulative slice size and distance (in dynamic instructions) showing regions with no contribution.
Additional Observations

**FALSE CONTROL DEPENDENCES:**

*control dependent regions are part of slice, but all paths from the branch contribute to the slice equivalently.*

**COMMON CASE: CONDITIONAL FUNCTION CALL**

\[
\text{INT A = \ldots;}
\]

\[
\text{IF (B) \{}
\]

\[
\text{ \quad FUNCTION();}
\]

\[
\text{\}}
\]

\[
\text{IF (A) \{}
\]

\[
\text{SAVE A;}
\]

\[
\text{\quad \ldots}
\]

\[
\text{\quad RESTORE A;}
\]

**OTHER CASE: CODE REPLICATION**

Currently refining infrastructure to handle these cases
Summary

**Pre-execution:** General technique for handling “problem” instructions
- Use the program to predict the program
- Requires small, accurate slices

**Speculative slices:**
- Exploit common-case behavior to reduce slice size while maintaining accuracy
- Some slices can be reduced to less than 10% of the 512 instructions preceding the criterion while maintaining greater than 95% accuracy
  - Much future work to be done
- Requires sophisticated profile information